ShareIff: A Sticky Policy Middleware for Self-Destructing Messages in Android Applications

António José Monteiro de Oliveira Goulão

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. José Manuel da Costa Alves Marques
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Ricardo Jorge Fernandes Chaves

May 2016
Acknowledgements

I would like to address my special thanks to my advisor Professor Nuno Santos for his tireless persistence, availability, guidance and the opportunity to do this thesis and give my contribute to the development of safer mobile operating systems.

I extend my thanks to Nuno Duarte for his support and advise during the execution of this work and to Miguel Costa.

A special thanks to my family and friends for their support and motivation to finish one more phase in my life.

Lisbon, May 2016
António José Monteiro de Oliveira Goulão
For my family, for my advisor and my friends,
A aplicação móvel Snapchat conseguiu um ganho imenso de popularidade ao introduzir um serviço inovador de troca de imagens utilizando mensagens que se autodestroem. Contudo, o Snapchat tem mostrado como é desafiante construir este tipo de serviços para as plataformas móveis atuais. Na verdade, seja por erros de programação ou devido às limitações dos sistemas operativos móveis existentes, no Snapchat e em aplicações móveis semelhantes, é possível recuperar mensagens supostamente apagadas, indo contra as expectativas dos remetentes, deixando assim milhões de utilizadores vulneráveis a violações de privacidade. Esta dissertação propõe o ShareIff, um middleware para Android que fornece uma API para troca e visualização segura de mensagens autodestrutivas. Com este middleware, o Snapchat, ou qualquer outra aplicação móvel semelhante, consegue codificar a mensagem a enviar no dispositivo do remetente e enviá-la para o recetor para que apenas o recetor possa descodificar a mensagem e visualizá-la de forma segura no seu dispositivo, dentro do intervalo de tempo especificado pelo remetente. O ShareIff fornece esta propriedade usando protocolos de criptografia especializados e mecanismos do sistema operativo. O ShareIff oferece uma interface de programação simples aos programadores de aplicações móveis adicionando uma sobrecarga marginal tanto ao sistema operativo, como às aplicações.
Abstract

Snapchat is a mobile application that got immense popularity due to an innovative photo sharing service that allows the delivery of self-destructing messages. However, Snapchat’s history has shown that building such services on modern mobile platforms is very challenging. In fact, either caused by programming errors or due to the limitations of existing mobile operating systems, in Snapchat and similar applications it is possible to recover supposedly deleted messages against the senders’ expectations, therefore leaving millions of users potentially vulnerable to privacy breaches. This dissertation presents ShareIff, a middleware for Android that provides an API for secure sharing and display of self-destructing messages. Using this middleware, Snapchat, or any similar application, is able to encrypt the message on the sender’s endpoint and send it to the recipient such that the message can be decrypted and securely displayed only on the recipient’s device for the amount of time specified by the sender. ShareIff provides this property by relying on specialized cryptographic protocols and operating system mechanisms. ShareIff offers application developers a simple programming abstraction and adds marginal overheads to both the operation system and applications.
Keywords

Self-Destructing Messages
Sticky Policies
Mobile Applications
Security and Privacy
Android
## Contents

1 Introduction ............................................. 1
   1.1 Motivation ......................................... 1
   1.2 Requirements ...................................... 3
   1.3 Contributions ...................................... 3
   1.4 Results ........................................... 4
   1.5 Document Structure ................................ 4

2 Background & Related Work ............................... 5
   2.1 Background ......................................... 5
       2.1.1 Overview of Snapchat ......................... 5
       2.1.2 Security limitations of Snapchat ............. 6
       2.1.3 Goals, threat model, and assumptions ..... 8
   2.2 Related Work ....................................... 10
       2.2.1 Self-Destructing Messaging Apps ............ 10
       2.2.2 Access Data Control Mechanisms ............ 11
       2.2.3 Information Flow Control .................... 12
       2.2.4 Secure Data Deletion ......................... 15
       2.2.5 Digital Rights Management .................... 18
   2.3 Summary ........................................... 21

3 Proposed Solution ...................................... 23
   3.1 Architecture ....................................... 23
   3.2 The Trustview API Abstraction .................... 24
   3.3 Security Protocols ................................ 26
List of Figures

2.1 Architecture of Snapchat. .................................................. 6
2.2 TaintDroid architecture. .................................................. 13
2.3 Android DRM architecture diagram. ................................. 19

3.1 Architecture of the ShareIff middleware. ......................... 23
3.2 Example of a trustview in a screen layout. ....................... 25
3.3 Examples of screening sessions. ........................................ 30
3.4 ShareIff overall architecture. .......................................... 32
3.5 ShareIff image rendering. ............................................... 33

4.1 Sharing keys between Photobrake users. ......................... 40

5.1 Execution time of envelope creation. .............................. 44
5.2 Android native and ShareIff’s image rendering execution time comparison. . . . . 45
List of Tables

2.1 System categories summary ........................................ 21

5.1 Experiment image resolutions and file size. ..................... 44
Acronyms

A P I  Application Programming Interface
A E S  Advanced Encryption Standard
A O S P  Android Open Source Project
C B C  Cipher Block Chaining
D R M  Digital Rights Management
E C B  Electronic Codebook
I F C  Information Flow Control
I P C  Inter-Process Communication
M A C  Mandatory Access Control
M V C  Model-View Controller
O S  Operating System
R E S T  Representational State Transfer
T C B  Trusted Computing Base
U I D  Unique Identification
Introduction

Over the last years, the demand for sharing digital content with others has been increasing rapidly. Everyday, people share millions of messages, images, videos and other contents using their mobile devices. Apple started this trend with the introduction of the smartphone, a powerful and affordable mobile device that can easily connect to the Internet. Since the release of iPhone\(^1\) in June 2007 (Honan 2007), other manufacturers started to produce their own mobile devices equipped with specific operating systems. Quickly appeared the need to have an interoperable operating system that could work across devices from different manufacturers. Android 1.0, the first commercial version, was first introduced in September 2008 (Morrill 2008) to serve this need. Currently, the environment surrounding Android provides a powerful platform for creating apps and an open marketplace to distribute them. With more than 1 thousand million devices active by September 2013 and 1.5 thousand million apps downloaded from Google Play\(^2\) each month (Google 2015a), Android has proven to be the most popular mobile platform.

Despite the advanced security capabilities of mobile devices, users still have many limitations in keeping their personal data under control. This lack of control can result in security breaches and loss of privacy. In particular, existent mobile OSes do not provide the tools necessary for the content owner to control what he shared with a friend. From the moment that his friend receives the content, the friend can do whatever he intends with the content (e.g., take a screenshot, edit, create a local copy, share with others, etc.). Most of the times, sensitive information is leaked and exposed publicly against the will of the content owner (Lee 2014; Johansen 2014).

1.1 Motivation

Self-destructing digital messages have emerged as an attempt to mitigate the lack of privacy-protection when sharing security-sensitive content among friends. Although this concept has been explored in the past by the research community (Geambasu, Kohno, Levy, and Levy 2009), self-destructing digital messages have become truly widespread by Snapchat\(^3\). Snapchat is a mobile application that allows users to share photos that “self-destruct” a few seconds after

\(^1\)https://www.apple.com/iphone/
\(^2\)https://play.google.com/store
\(^3\)https://www.snapchat.com
being viewed by the recipients. A sender can take a picture (e.g., a photo from the camera) and share it with a friend specifying a visibility timeout, which by default is 10 seconds. When the recipient opens the picture – named *snap* – Snapchat displays it on the recipient’s screen for the duration of the visibility timeout. As soon as the timeout expires, the picture vanishes from the screen and cannot be read again by the recipient. Since 2011, the year of Snapchat’s first release, the service has been upgraded to support other kinds of content sharing (e.g., videos, and text) and richer sharing policies (e.g., replayed for three times before permanent deletion). As of 2014, Snapchat had grown incredibly popular, reaching a user base of 100 million users and a content exchange rate of 700 million photos and videos per day (Insider 2015).

However, in spite of all its success, Snapchat has been troubled by a history of security flaws, enabling supposedly deleted snaps to be recovered by receivers or even by untrusted third-parties. For example, in a high-profile data breach, named “the snappering”, over 100K snaps were leaked by an API exploit (Insider 2014). In many cases, security flaws come from application programming errors, ill-designed cryptographic protocols, or insecure RESTful APIs. In other cases, insecurity is due to existing limitations of Android or other mobile operating systems, which, for example, allow users to take screen shots while snaps are being displayed. Security flaws have motivated a formal complaint to the US Federal Trade Commission (of America Federal Trade Commission 2014), forcing Snapchat to modify its privacy policy to clearly state that: “We can’t guarantee that messages and corresponding metadata will be deleted within a specific timeframe” (Snapchat 2015).

Nevertheless, Snapchat remains a highly popular application, which makes us wonder whether security is important at all for its users. In 2014, researchers from the University of Washington conducted a user survey (Roesner, Gill, and Kohno 2014) to help understand how people use Snapchat. This study shows that for most respondents, security is not the main concern: they enjoy the fact that messages “disappear”, but may not need messages to disappear securely. However, while it appears that security is unimportant, up to 25% of the respondents admitted to having sent sensitive content: sexual content, legally questionable material, private documents, and insulting or offensive messages. Moreover, a non-negligible fraction (38.6%) reported that, in response to learning that message destruction is not secure, their behavior would have been changed, e.g., by avoiding using Snapchat, sending different content, or sending messages to different people. For such users, a security breach could bring serious and irreversible consequences. Given the large size of the Snapchat user base, millions of users could be affected. Therefore, this study suggests that there is room for more secure self-destructing messaging applications.
1.2 Requirements

We intend to provide application developers the necessary mechanisms to enforce the security of digital content, such as photos, as well as provide a mean to the users to have more control over the contents they share, while imposing a low overhead to the system and app executions. These are the main requirements:

1. The solution should provide security to the content being shared, while on transit and on the device. It should be impossible for an attacker to see the content inside the package being transmitted through the network, and impossible to modify that content without being noticed. The solution should provide integrity, authenticity, and confidentiality;

2. The solution should be efficient by imposing a low overhead to the system and to the application that is using the solution. The solution should not impair the user interaction with the application and the device.

3. The solution should have an API that does not significantly increases the programming complexity. The new functionalities should be of easy access and quick understanding.

1.3 Contributions

To support the development of secure self-destructing mobile applications, this dissertation presents ShareIf, a middleware for Android platforms. ShareIf provides an API that allows mobile applications to encrypt a message on the sender’s endpoint and send it to a recipient such that the message can be decrypted and securely displayed only on the recipient’s device for the amount of time specified by the sender. ShareIf ensures that the message (1) cannot be recovered while traveling from the sender to the receiver, (2) cannot be digitally captured while it is being displayed on the receiver’s device, and (3) is effectively deleted after the visibility timeout elapses.

To enforce remote message self-destruction, ShareIf adopts a sticky policy architecture. At the sender side, the message is enclosed inside a cryptographic envelope along with a sticky policy, which specifies the visibility timeout to be enforced at the recipient endpoint. To prevent interception or modification of the message in transit, the envelope can be decrypted only with a key that is maintained by a protected ShareIf service residing in the recipient’s device. While displaying the message, ShareIf keeps the raw bytes of the message inaccessible to the application and to potentially dangerous OS functions that could cause data leakage (e.g., screenshot capture). To guarantee that the message cannot be recovered after deletion, ShareIf ensures correct timer countdown, sanitizes internal buffers, and wipes envelope keys. ShareIf offers
its services to application programmers through a simple primitive named TrustView, which is inspired by a programming abstraction familiar to Android user interface designers.

Since ShareIff relies on the security of cryptographic algorithms and on the integrity of the recipient’s operating system, the operating system must be trusted. Bootstrapping trust in the OS can be achieved by implementing trusted boot using trusted computing hardware (Group 2006). Note that ShareIff is not intended to prevent analog side-channel attacks, i.e., take a photo of the screen using an external camera while the message is on display.

We implemented ShareIff and tested it on Nexus devices. Our evaluation shows that ShareIff adds negligible performance overheads to the system and most of its cost comes from encrypting and decrypting snaps. To validate the programming effort of ShareIff, we implemented a simple photo sharing application based on trustviews. Our experience is that ShareIff is very simple to use, requiring just a few lines of code in order to incorporate into the application.

1.4 Results

The results of this work can be enumerated as follows:

1. Design of a middleware to enforce self-destructing messages in mobile applications.
2. Implementation of prototype for the Android platform which demonstrates the effectiveness and viability of our design.
3. Experimental evaluation of this system on a real hardware testbed.
4. Implementation of a use case application that leverages the developed middleware to enable secure photo-sharing.

1.5 Document Structure

The rest of this document is organized as follows. In Section 2, we present some background on Snapchat and discuss the related work on privacy protection on mobile devices. Then, Section 3 presents our solution to enable the development of applications that enforce remote self-destruction of digital messages (including photos). Section 4 describes the implementation of our system, and Section 5 presents the evaluation of the system. Finally, in Section 6, we present our conclusions and discuss future work.
This chapter is divided in two main parts. Section 2.1 provides some necessary background information to understand the motivation and goals of this thesis. It starts by providing an overview of Snapchat and discusses the security limitations of this popular self-destructing message application. Then, it clarifies the goals of this thesis, as well as the threat model and assumptions we make to devise our solution. Section 2.2 discusses the related work.

### 2.1 Background

Snapchat is a mobile application that pioneered in implementing a remote photo sharing service with remote photo self-destructing capability. We use Snapchat as a representative example to motivate the need for middleware that can enhance the security of such applications. Designing such middleware is the main focus of this dissertation.

#### 2.1.1 Overview of Snapchat

Snapchat was first launched in September of 2011 and right now has more than 100 million installs on Google play store. Snapchat is also available for iOS. It allows users to share pictures and videos ephemerally (snaps), and send messages. The sender can choose a time interval between 1 and 10 seconds and the users with he wants to share the picture with. There’s also an option to share pictures with all the users for 24h.

Snapchat follows a typical client-server architecture. As shown in Figure 2.1, servers are responsible for managing the user base and for forwarding snaps between clients. Clients consist of instances of the Snapchat mobile application running on users’ devices. From this application, users can send or receive snaps to / from their friends. Clients communicate with the servers over HTTP through a REST API.

To send a snap to her friend, Alice—the sender—selects a picture, picks Bob’s contact from her friend list, and defines the visibility time of the snap (e.g., 10 seconds). Alice clicks the send button, and the local client uploads the snap to its servers, triggering a notification to Bob’s client that a new snap is awaiting to be read. When Bob—the recipient—opens the snap, the local client downloads the snap from the servers and displays it on the screen. Once the
visibility timeout expires, the image vanishes from the screen and is deleted from the local client. The snap is removed from the servers after being downloaded. Until a snap is not opened by a recipient, it remains in Snapchat’s servers for 30 days, whereupon it is permanently deleted.

Just like in any other mobile application, the Snapchat client relies on the security mechanisms provided by Android: *application sandboxing* and *permissions*. Android has a Linux based kernel that enforces UID-based application sandboxing by forcing apps to run inside individual processes. Applications may write persistent data to their private directories and leverage application sandboxing to prevent unauthorized access by other applications. Additionally, Android provides a permission model which it leverages to regulate app access to OS resources. This model is also used to protect apps’ resources from other app unauthorized accesses.

### 2.1.2 Security limitations of Snapchat

Unfortunately, Snapchat has had security limitations that may result in unauthorized access to users’ snaps. Unauthorized access occurs in two cases: (1) a snap can be retrieved by someone that is not in the recipient list (L1, L2), or (2) a snap can be recovered by the recipient violating the visualization time constraints specified by the sender (L4, L5, L6).

**L1. The “analog hole”:** Consists of side-channel attacks aimed at obtaining an analog copy of a snap while it is displayed on the receiver’s device. To mount such attacks and capture the screen output, the receiver requires auxiliary external hardware, for example an external camera, and there is normally loss of quality. Snapchat can neither prevent nor detect such attacks.

**L2. Built-in screenshot function:** An easy way for a user to make a persistent copy of a snap is by using Android’s built-in screenshot service. A screenshot can be taken, e.g., by holding down the “volume down” and the “power” buttons for 1-2 seconds. Since Android applications
cannot block screenshot capture, Snapchat cannot prevent the user from obtaining a digital copy of the snap and save it on the phone’s persistent memory. To alleviate this problem, Snapchat leverages an Android function to notify the sender if the receiver has taken a snapshot of the picture. However, this mechanism can easily be thwarted if the receiver disconnects its client from the network before taking the screenshot (iDownloadBlog 2014). This move prevents the receiver’s client from forwarding the notification to the Snapchat server.

L3. Persistent buffers: After downloading a snap from the servers, Snapchat keeps a local copy of the snap on the phone’s persistent memory. In Snapchat’s first version, such a copy was saved in unencrypted format outside its application sandbox, i.e., in a public space (Leavitt 2014). This flaw allowed for a rogue application to easily access it and create a persistent copy of the snap. The user could also plug the device into a computer over a USB cable and extract the snap from the device using a debugger. Access to snaps was further facilitated by the fact that Snapchat did not delete the snap’s local copy after the time limit expired. On later versions, Snapchat fixed this problem by making sure to keep any snap copies on its private area and in encrypted format. (Ironically, content can be decrypted due to a flaw in the use of encryption.)

L4. Flawed use of encryption: The first version of Snapchat did not use encryption at all. The snaps travelled the network unencrypted and were stored in Snapchat servers in plaintext. As a result, a snap could be freely accessed by a network eavesdropper or by someone with privileged access to the server. In an effort to secure the snaps while in transit between sender and recipients, Snapchat’s later versions started using AES for end-to-end encryption of snaps between sender and receiver endpoints. However, the implementation was seriously flawed. Firstly, AES used ECB, an insecure block encryption mode. Secondly, and most critically, the symmetric key for encrypting and decrypting the content is unique—for every piece of content, for every user—and is provisioned directly in the Snapchat’s application binary. By reverse engineering the binary, researchers were able to recover the encryption key, which is now public domain (Caudill 2012b). Recent versions of Snapchat fixed the encryption mode to CBC and use two encryption keys. However, both keys remain hardcoded in the application binary and have been reversed engineered (GibsonSec 2013). As a result, Snapchat remains insecure.

L5. Insecure APIs: Whenever a snap is rendered on the receiver’s device, the responsibility for limiting the exposure time of the image is of the Snapchat mobile application. Therefore, a third-party application that can impersonate the authentic Snapchat application and successfully download the snap from the server can easily bypass the client-side protections and store a permanent copy of the snap. To prevent such attacks, Snapchat servers expose a private REST API that includes defense mechanisms against third-party applications. However, in early versions of Snapchat such mechanisms were very fragile. They relied upon obscure security protocols encoded in Snapchat’s mobile application. By reverse engineering the appli-
cation binary, researchers managed to decode the API protocols (Caudill 2012a), allowing for an ecosystem of tools (Lackner 2013) and third-party apps to appear (PhoneArena 2014). By installing such apps, it was straightforward for a recipient to save a snap and replay it any number of times. Although the latest Snapchat API version has been reinforced to mitigate third-party apps (Snapchat 2013), the fundamental problem remains: the security of the service depends upon the obscurity of the API.

L6. Replay attacks: As described above, to open a snap on a recipient’s device the Snapchat client must perform a sequence of steps: download the content, decrypt it, render it on the screen, set a timer, etc. However, Snapchat cannot guarantee the atomicity of this sequence. As a result, by cleverly interrupting or stalling this process, a recipient can visualize a snap multiple times or for a longer amount of time than permitted. Such attacks work particularly well for videos, which can be replayed on the recipient’s device for an unbounded number of times by disconnecting the device from the network immediately after the video starts playing.

L7. Rooting the device: In order to properly implement visualization restrictions on the recipients’ devices, Snapchat requires that the underlying operating system and hardware platform operate correctly and that the user cannot gain arbitrary control over the software of the device, namely the OS. Such conditions are expected to be found on a typical commodity Android device. However, by rooting the device, a user can execute applications with superuser privileges, allowing such applications to access the Snapchat’s sandbox and create permanent copies of the downloaded snaps. Although the rooting operation involves some degree of sophistication, it is not uncommon for Android users to root their devices. Against such attacks, Snapchat is helpless.

2.1.3 Goals, threat model, and assumptions

From the security limitations presented above, we see that while some of them are fundamental (namely L1), others are caused by deficiencies in the design or implementation of the Snapchat application (L3, L4, L5, L6) or by existing limitations in the Android operating system (L2, L7). Our goal is to enable the development of secure self-destructing message sharing applications. In particular, our focus is on mitigating the security limitations due to deficient application programming or to OS-related issues. We aim to provide a general solution for self-destructing message delivery, which can be used not just by Snapchat, but also by alternative applications, such as Cyber Dust\(^1\) or Confide\(^2\). Note, however, that it is not our goal to mitigate potential side-channel breaches through the “analog hole” (L1).

\(^1\)https://www.cyberdust.com
\(^2\)https://getconfide.com
2.1. BACKGROUND

To attain our goals our approach is to build a middleware based on a sticky policy architecture. Essentially, the middleware aims to provide a simple primitive to applications that enables senders to encode self-destructing messages into cryptographic envelopes. Each envelope includes a sticky policy which defines the visualization timeout. The middleware must ensure that the envelope can only be opened at the receiver’s endpoint and that the enclosed message (1) cannot be recovered while traveling from the sender to the receiver, (2) cannot be digitally captured while it is being displayed on the receiver’s device, and (3) is effectively deleted after the visibility timeout elapses. The middleware must provide simple programming abstractions to application developers.

We contemplate the following classes of attacks:

- **Network attacks**: An attacker may intercept the communication, collect envelope packages, modify them, and inject new ones. An attacker may attempt to impersonate legitimate receivers in order to launch MITM attacks.

- **API exploits by third-party applications**: Third party applications may attempt to retrieve envelope data from the application server API or from the public interface of the application sandbox on the recipient’s device.

- **Remote application exploits**: The application may have bugs that can be exploited by an attacker on the recipient’s endpoint and result in unwanted message recovery.

- **Malicious user operations**: A message recipient may try to use operating system services, such as taking screenshots or attaching a debugger, in order to create digital copies of messages.

- **Forensic analysis of persistent memory**: A forensic analyst with physical access to the device may be able to recover relevant material from devices’ persistent memory (e.g., key material, encrypted or unencrypted envelopes, etc).

We assume that the operating system’s kernel and middleware services are trusted and are therefore part of the trusted computing base. To prevent attacks based on device rooting (L7), it is possible to employ hardening techniques coupled with trusted computing hardware in order to assure trusted boot state (ARM 2009). Such techniques can be deployed by device manufacturers when manufacturing ShareIff-enabled devices. We also assume that ShareIff devices can be shipped with a cryptographic key pair that can be used to uniquely identify the device. We also rely on the fact that the content of the device’s volatile memory is lost when a device is powered down in order for the data to be irrecoverable when the device and the applications are restarted.
2.2 Related Work

Our work is related with prior work from different areas. In this section, we provide an overview of related work in the following topics: self-destructing messaging apps (Section 2.2.1), Information Flow Control (Section 2.2.3), secure data deletion (Section 2.2.4), and Digital Rights Management (Section 2.2.5).

2.2.1 Self-Destructing Messaging Apps

Although Snapchat is the most popular application of its kind, there are other mobile apps in the market focused on enabling data sharing with remote self-destruct capability. There are several applications that allow information to be shared ephemeraly or limit the way the information is used. We provide an overview of the most relevant of such applications below:

**Cyber Dust**\(^3\), launched in 2013 for the iOS, is now available for Android and Windows Phone. Its goal is to avoid that any user that communicates with you, uses your messages against you. In simple words, all the messages are destructed after thirty seconds. During data transmission all data is fully encrypted and when this is on the device, is not written to permanent storage, it’s kept in the device memory. Still, this application has some limitations for our use case, like not preventing screen shots or not allowing the users to save pictures they receive.

**Confide**\(^4\) is an application that allows users to text privately. What this application has of interesting is the way it prevents screen shots. The text of the message is covered and it is only displayed word by word when the user slides the finger over the text. This way screen shots of the entire message are impossible as only one word is visible at a time and when the first screen shot is detected, the entire message is erased. Although Confide does not support sharing any type of media, neither it can prevent side channel attacks.

**Cluster**\(^5\) is an application that allows sharing pictures inside small groups defined by the users or with only some of his friends. It has the option to see who viewed the picture and who liked it. It has the limitations of letting the users save the picture even if the user who shared it doesn’t intend to, and take screen shots.

**Bonfyre**\(^6\) is similar do Cluster but the owner of the group has the power of controlling the possible actions of the members inside the group, by assigning specific permissions to each user of the group.

---

\(^3\)http://www.cyberdust.com/
\(^4\)https://getconfide.com/
\(^5\)https://cluster.co/
\(^6\)https://www.bonfyreapp.com/
2.2. RELATED WORK

**Sicher**\(^7\) is a private messaging application capable of sharing files like PDFs, pictures, videos, documents and others. With end-to-end encryption, private push notifications, group chats, and self-destruction options for both messages and files sent.

All these applications try to enforce remote message self-destruction, but they all have similar limitations to Snapchat due to the lack of support from the underlying system (see Section 2.1.2). There are other applications that allow the user to share its pictures with anyone he wants, but they don’t give the user so much control over the content after being shared. This is the case of Instagram\(^8\), launched in October 2010, it has more than 300 million monthly active users. Instagram allows sharing pictures and videos with everyone within the user’s group of followers and with every user in the platform by using hashtags in the description of pictures. Instagram doesn’t prevent screen shots, and doesn’t allow users to save the pictures or videos they receive.

### 2.2.2 Access Data Control Mechanisms

Access data control can be implemented in two ways: or the policies are defined on the server storing the data and is the server that checks if the requester can access this data, or they accompany the data everywhere and are enforced on the receiving device. Both types of implementations rely on Mandatory Access Control (MAC). In mobile operating systems, the mechanisms that enforce MAC are implemented on the middleware layer, at the kernel layer or both. Most of them are tailored to address a specific problem, like controlling accesses to user’s private data or securing the platform integrity.

**FlaskDroid** (Bugiel, Heuser, and Sadeghi 2013), and similar other systems, provide a security framework for accessing sensitive data. FlaskDroid provides mandatory access control simultaneously on Android’s middleware and kernel layers. FlaskDroid is based on the Flask security architecture (Spencer, Smalley, Loscocco, Hibler, and Andersen 1999). It implements a user-space security server which is responsible for assigning security context to objects such as files or IPC and middleware resources. When an application tries to access a resource, first the application queries the security server, and if the application is allowed to access the object, it continues executing. On the kernel layer, FlaskDroid employs Security Enhanced Linux (Google 2015c) in Android (SE Android). SE Android is essential to prevent malicious applications from circumventing the user-space security server by escalating their privileges, complementing the policy enforcement at the middleware layer.

**Hails** (Giffin, Levy, Stefan, Terei, Mazieres, Mitchell, and Russo 2012) is an alternative system that makes use of model-view-controller (MVC) architecture and modifies it to model-policy-

---

\(^7\)http://www.shape.ag/en/index.php

\(^8\)http://instagram.com/
view-controller (MPVC) architecture, introducing access data policies in the model and removing every policy code that might exist in View or Controller. Hails is a web platform framework where security policies are specified in a single module creating a mandatory access control point to resources stored by the model module. These policies are defined for a collection of documents, for each document and each field of the document at the server keeping the data and can be composed, restricting the access to the finer-grained data. To achieve their goals, the authors break apart the architecture leaving Models-Policies (MPs) on the trusted side and Views-Controllers (VCs) on the untrusted side. MPs are responsible for keeping the user data secure and enforce the policies the user defined. The VPs are also responsible for enforcing the policies, but they may not be trusted. With this architecture, policies follow data through the system allowing them to be enforced globally. For instance, when a user requests data from the server, this first checks if the user requesting the data has permission to do it, and only if it has, it sends only the data that this user can see.

**Sticky policies** (Pearson and Mont 2011; Trabelsi, Sendor, and Reinicke 2011; Karjoth, Schunter, and Waidner 2003) on the other way, are attached to the data and follows it to wherever it goes. They are similar to how digital rights management (DRM) attach copyright policies to content. The access to the content is only allowed upon satisfaction of the policies that accompany it. When an application requests a content, first the policy enforcement component checks if the application or user has the right to fetch the content, and only if it pass that check, the content is retrieved to application. Usually the data is always encrypted and is only decrypted upon acceptance and satisfaction of the constraints the policies impose. There are various techniques using different encryption mechanisms that provide sticky-policy protection of data. The most known are: public-key encryption (PKE) (Bellare, Desai, Pointcheval, and Rogaway 1998) and Identity-Based Encryption (IBE) (Shamir 1985). There is also attribute-base encryption (ABE) (Sahai and Waters 2005) and Proxy Re-Encryption (PRE) (Mambo and Okamoto 1997).

If we reason about these systems, we realize that the MPVC is not the best approach to our system since we don’t want to rely on trusted servers as the point of resource access control, and a better approach would be to use sticky policies to enforce on the receiver of the information, the policies defined by the sender.

### 2.2.3 Information Flow Control

Information flow control (IFC) is a technique aimed at keeping track of how security-sensitive information flows throughout the execution of a program. In this section, we discuss two relevant IFC systems for mobile platforms.
2.2. RELATED WORK

TaintDroid (Enck, Cox, Jung, Gilbert, gon Chun, McDaniel, and Sheth 2010), a pioneering framework, introduced the tracking of tainted data from variables to file-level information inside the device, successfully detecting unauthorized information leakage. TaintDroid is a modification to the Android firmware and is mostly concerned in analyzing how apps handle privacy sensitive data (e.g., location, phone IDs, camera, etc.) and monitor when this data leaves the phone.

TaintDroid tracks data flow through third-party applications, assuming that these are not trusted. It labels (taints) the data from privacy-sensitive sources (e.g., the location provider) and applies labels as sensitive data propagates through application variables, files or IPC during its execution. When sensitive data leaves the system, TaintDroid logs its labels, the application responsible for transmitting the data, and its destination. This approach also reveals the level of granularity used in TaintDroid. To achieve this level of granularity, the authors had to modify the Dalvik VM interpreter to store and propagate taint tags on local variables and arguments, class fields, and arrays. For local variables and arguments, TaintDroid stores the tags in the adjacent memory position of the internal execution stack. For class fields it uses a similar approach, storing the taint tags adjacent to the memory position inside static and instance field heap objects. For arrays, it stores only one taint tag per array to minimize the overhead it could cause in large arrays.

Figure 2.2 provides an example of how data flows within TaintDroid. A trusted application reads the International Mobile Station Equipment Identity (IMEI) of the mobile device in 1. In 3 we depict a taint tag moving from one variable to another and being sent through Binder to
another application on 5. In 9 the value of the variable is sent to the Taint Sink that might be the Java library that writes to the network. TaintDroid tracks data from a variable to a file, leaving a most wanted level of abstraction aside, the application-level data object.

With respect to Java Native Interface (JNI) methods, TaintDroid doesn’t monitor any code. Instead, it uses manual instrumentation, heuristics, and methods profiles to achieve the same semantics described above. Manual instrumentation was mainly used on internal VM methods. This methods are called by passing a pointer to an array of 32-bit register arguments and a pointer to a return value. These are the values that are tagged by TaintDroid. For JNI methods invoked through the JNI call bridge, the authors implemented a method profile table for tag propagation. A method profile is a list of (from, to) pairs that indicate flows between variables. This table is consulted every time a JNI method returns, tagging every return. Additionally, the authors included a propagation heuristic patch for JNI methods that operate with primitive and String arguments and return values. The heuristic assigns the union of the method argument taint tags to the taint tag of the return value. This heuristic covers many existing methods but still has false negatives when used with objects that are not of type primitive nor String.

When exchanging data between applications, TaintDroid inserts a taint tag to each message passed through the IPC. This taint tag represents the upper bound of the taint marks assigned to variables contained in the message. This approach helps minimize performance and storage overhead and also prevents applications from removing taint tags. As with arrays, having all data items with the same tag leads to false positives. For secondary storage, each file has one taint tag which is updated on file write, using the file system extended attributes, and is propagated to data on file read. As with arrays and IPC, this approach also leads to false positives.

**Pebbles** (Spahn, Bell, Lee, Bhamidipati, Geambasu, and Kaiser 2014) is a related system that addresses the problem of tracking an application-level data object in the Android OS. In Android, an object can be stored in a database, an XML file or in the filesystem with its own format. Pebbles plugs into this storage abstractions and extracts the information needed to construct logical data objects (LDOs). An LDO is composed of nodes. Each node is a database row, an XML item or a raw file (e.g., photo). A device-wide globally-unique identifier (GUID) is assigned to each data item and stored with them. Each node may belong to other node (e.g., the photo belongs to the email), making an LDO. An LDO is a directed graph where nodes can be addressed from one root node.

Pebbles modifies Android OS in three ways:

- It plugs into the storage abstractions to obtain the persisted objects and follows references to them to obtain relations between them;
- It uses a modified version of TaintDroid to track data flows and discover relationships. Each GUID is a taint mark used to track the object in memory;
2.2. RELATED WORK

- It introduces the Pebbles Object Manager which records all the relationships and creates the LDOs. The Pebbles Object Manager also exposes the LDOs to Pebbles-based data management tools.

The Pebbles framework provides new tools to increase the user visibility and control over data management and new applications can be built on top of Pebbles.

In summary, information tracking systems are very useful as a basis for other frameworks that allow other systems to enforce object deletion, object access auditing and object access restriction such as CleanOS (Tang, Ames, Bhamidipati, Bijlani, Geambasu, and Sarda 2012) which is based on TaintDroid. Pebbles approaches from what we require with this work, by tracking the application objects inside the mobile device, but it has no ability to track it outside of it.

2.2.4 Secure Data Deletion

There is a number of relevant systems concerned about deleting data in a computer system. First, we present two systems concerned about deleting any execution trace on computers. Then we introduce systems to protect data against device loss or theft, and systems designed to delete sensitive data when stored remotely from the device.

Lacuna (Dunn, Lee, Jana, Kim, Silberstein, Xu, Shmatikov, and Witchel 2012) is a system whose goal is to make irrecoverable, for a non-concurrent attacker, any state generated during program execution after this has been terminated. To achieve this goal, Lacuna executes applications in private sessions in a virtual machine (VM) under a modified QEMU-KVM hypervisor on a modified host operating system (Linux). By running applications in a VM, Lacuna confines inter-process communications. However, applications need to communicate with peripheral devices (e.g. GPU), and by doing this, data could be leaked into kernel memories or shared memory, making the erasing of this leaks after application termination very difficult or impossible. To prevent this, Lacuna implements ephemeral channels.

Ephemeral channels are private connections between the VM and the hardware so that only the endpoints see the private data. Lacuna uses two kinds of ephemeral channels: hardware channels and encrypted channels. Hardware channels exploit virtualization support in hardware. To get exclusive control of the hardware, Lacuna uses peripheral component interconnect (PCI) device assignment. This way, sensitive data passed between the VM and the hardware never leaves traces in the host. Encrypted channels use encryption with standard key exchange to establish a trusted channel. Data is encrypted in Lacuna’s virtual machine monitor and is decrypted in a small software proxy developed for each piece of hardware. This proxy is located as close as possible from the hardware. When the private session is terminated, the OS and the
proxies memories are zeroed by themselves and the key used to encrypt the communication is erased, making any data left unrecoverable.

Lacuna allows to run any operating system and any application on top of the VM without need of any modification. Furthermore, users can perform sensitive tasks, within a private session, in parallel with non-private task. If the user is not concerned if a task leaves traces on the computer he is using, Lacuna executes the non-private application directly on the host OS and does not impose any performance overhead to the non-private application.

**PrivExec** *(Onarıloğlu, Mulliner, Robertson, and Kirda 2013)* is a generic Linux OS that provides private execution as a service, by creating a logical distinction between public and private processes. Its main goal is to guarantee that every application can execute in a mode where any write made to the filesystem or swap, will not be recoverable during or after application execution. Each application may have more than one process, thus when executing in private mode, every process belongs to the same private process group.

To achieve this goal, PrivExec randomly creates an ephemeral private execution key (PEK) on process group creation and assigns the PEK to the process group. This key is used to encrypt all data produced by the private process group and is never exposed to the user or any other process, neither written to local storage. The PEK is securely wiped by PriveExec on process termination. For each process group there is a secure storage container which lives only during process execution and can only be accessed by the private process group. Private process groups can read and write from and to the secure storage, and can only read from the public filesystem. If a process tries to modify an existing file on the public filesystem, PriveExec creates a local copy on the secure storage and all reads and writes are redirected to the new file.

As for the filesystem, PriveExec also protects private process’ memory pages that swapped to disk. Each page is encrypted with the process’ PEK before being written to disk, imposing a per-application partition of the swap space. In addition to the private storage mechanisms, PriveExec also ensures that private processes can communicate with other processes from the same group, but not without unrelated processes. Public processes can write data to private processes, but the opposite doesn’t happen.

**CleanOS** *(Tang, Ames, Bhamidipati, Bijlani, Geambasu, and Sarda 2012)* is another relevant system concerned about secure data deletion. As previous systems are used on desktop computers, CleanOS is specific for mobile devices. CleanOS is an Android-based OS that manages sensitive data rigorously in order to prevent data from being exposed after loss or theft. It uses a trusted cloud to prevent sensitive data from falling in the wrong hands.

CleanOS introduces two major modifications to Android to achieve its goal. First it introduces the concept of sensitive data objects (SDOs), and second, introduces a new type of garbage collector (GC), the evict-idle GC (eiGC). An SDO is a collection of Java objects, files,
and database items, used by applications to manage their sensitive data. When an application creates an SDO, the application creates a description of the content associated with the SDO, assigns a unique identifier within the context of the application, assigns an encryption key, and finally registers it with the cloud database. One application can have multiple SDOs. Each SDO identifier is going to be used as a taint mark for each object belonging to the SDO. CleanOS uses a modified version of TaintDroid to track this objects. Whenever an application accesses an object from a SDO, the timing of the access is recorded for that SDO. The eiGC monitors when the objects are accessed and evicts objects from RAM, even if they are accessible by other objects, after a certain period of time. To evict the SDO, the eiGC encrypts every data field of every object belonging to the SDO with a key derived from the encryption key of the SDO. After SDO has been evicted, the encryption keys are securely destroyed.

The authors also modified the Dalvik VM to handle objects faults when these were previously evicted by the eiGC. When an application tries to access an evicted object, Dalvik fetches the encryption key of SDO from the cloud database, it generates the key used to encrypt the data fields of the objects of the SDO, and decrypts the data. The encryption key of the SDO is cached in the Dalvik VM and is securely destroyed when the SDO is again evicted. All this process is performed while the objects are in RAM, but CleanOS also evicts files and database items. Every file and database item is also tainted with the SDO identifier to keep track of it. Thus, before data is written to filesystem or database or when the data is retrieved from the filesystem or database, the data is encrypted or decrypted using the same process described above.

This process works well when the device is connected to a network. To handle disconnected operation, the authors of CleanOS came with two solutions. First, for short-term disconnection, the time to evict already existent SDOs can be extended, and second, for long-term disconnection, the encryption keys of the SDOs are hoarded in the device.

Vanish (Geambasu, Kohno, Levy, and Levy 2009) has the goal of destroying sensitive data in a secure way after some time, without any explicit intervention from the user after this data has been published, without using secure hardware, and without using any centralized system, preventing a non-concurrent attacker from obtaining a copy of it, even if the attacker got it before its expiration.

To achieve this goal the authors present the notion of a vanishing data object (VDO). A VDO is a data object which contains the encrypted data, a random access key used to derive the indexes of the nodes where bits of the key were stored and can be retrieved from, the number of nodes containing parts of the key, and a threshold, the number of nodes needed to reconstruct the key. When constructing a VDO, the user already has an idea of the period of time the data is going to be available and passes this period to Vanish with the data.

The list of indexes derived from the random access key included in the VDO, index nodes
in a peer-to-peer network. Vanish leverages one or more existing, decentralized, large-scale Distributed Hash Tables (DHTs), like Vuze\(^9\). Each node of the DHT indexed contains part of the key used to encrypt the data. To encrypt data, Vanish uses a random data key to encrypt the data. This key is then divided between a number of nodes, chosen pseudo-randomly from the DHT network. The key is then destructed locally. To decrypt data, Vanish does the opposite process. It gets the locations of the parts of the key, accesses the nodes to retrieve the shares, reconstructs the key and decrypts the sensitive data. Prior to timeout, the object can be accessed at all time by the receiver, which only has to pick all the parts necessary to reconstruct the key and decrypt the data. After some time it becomes impossible to recover the entire key and decrypt the data due to the churn of the DHT network. If the time during which data is available is insufficient, Vanish provides a mechanism to refresh VDO shares in the DHT. Vanish retrieves the original data key before timeout and re-splits it among a new set of nodes.

**Xiao Fu et al.** (Fu, Wang, Wu, qi Yang, and zhao Wang 2014) present a system similar to Vanish but specifically targeting the email system. While in Vanish only the key is divided among the DHT nodes, in this system the authors split the data among several servers using the network-based data self-destruction method invented by Xiao Fu\(^10\). The sensitive data is ciphered with AES with a random key. The ciphered data is then divided into pieces with no more than 256 bytes. The key is disguised as one of the pieces by attaching a pseudo-random sequence to it. Each piece is generated an Universally Unique Identifier (UUID). A time of expire (TOE), defined by the user in minutes, is attached to the piece and everything is stored into the database. To access the data, users use a Self-Destructing Data (SDD) link were every UUID was combined as a string. Every piece can be accessed until one of the pieces expire.

### 2.2.5 Digital Rights Management

Digital Rights Management (DRM) aims to provide a secure way for content providers to distribute digital content (e.g. music, videos, images), preventing end users from copying, converting or modifying it. We present two existing solutions for DRM, one developed by the industry, the other by the research community.

**Android DRM Framework**\(^11\), shipped with the Android distribution since version 3.0, is depicted in Figure 2.3. This framework is implemented in two layers: the DRM Framework API and the DRM Manager. The first is exposed to applications and runs through the Dalvik VM. This framework lets applications manage digital content according to the attached license constraints. The second, written in native code, exposes its interface to DRM plug-ins to

---

\(^9\)http://www.vuze.com/
\(^11\)https://source.android.com/devices/drm.html
2.2. RELATED WORK

handle rights management and decryption for various DRM schemes. The DRM framework was designed to abstract the details of the specific DRM scheme implemented by a plug-in. The DRM Framework includes interfaces to decrypt the DRM content, register users and devices to online DRM services, and extract constraint license from the license associated to the digital content. A mobile device may have more than one DRM plug-in installed. To manage all the installed plug-ins, the DRM Manager has a Plugin Manager in charge of loading/unloading all the available plug-ins. Which DRM plug-ins a device has installed, is up to its manufacturer. Plug-ins like Widevine\textsuperscript{12} or Discretix OMA DRM\textsuperscript{13} are widely used. Due to the abstract nature of the Android DRM framework, the problems detected with other solutions (e.g. OMA DRM v1) (Chuang, Wang, and Lin 2010), a need for a complementary solution became evident.

**Porscha** (Ongtang, Butler, and McDaniel 2010) is an example of a content protection framework that enables the sources of contents to express security policies to ensure that these contents are sent to the target phones, processed by endorsed applications, and handled in the way the source intends to. Porscha is constrained to the mobile phone segment as it uses the mobile phone number (MSISDN) as public key identity. It extends the XML OMA policy with the intent of protecting the content when it is delivered to the phone and once it is in the phone. Unlike OMA that doesn’t protect the contents from being accessed by applications that are not the intended to use them.

Porscha enforces fine-grained content policies in three ways: 1) only the authorized mobile phone can access the digital content, 2) only applications approved by the content provider can

\textsuperscript{12}\url{http://www.widevine.com/}

\textsuperscript{13}\url{http://www.discretix.com/}
access the digital content, and 3) content might be subjected to use constraints (e.g., limited time to use, no sharing, etc.). For the first, Porscha extends OMA DRM to use the phone number as the identity of the device instead of the International Mobile Subscriber Identity (IMSI) or WAP Identity Module (WIM). On the second, Porscha extends OMA DRM by allowing the sender to specify which applications can use the digital content. The hash of the application signed with the developer key is used to identify the application. Finally, Poscha introduces the possibilities of specifying if the content can be read, modified or deleted by the receiving applications. This policies are enforced both in transit as on the device.

When on transit, Porscha uses identity-based encryption (IBE) to ensure confidentiality, integrity and authenticity of the digital contents. When the digital content arrives to the mobile device, Porscha uses the recipient’s phone number as public key to decrypt the digital content. For larger contents, e.g., email, Porscha generates a one-time 128bit AES symmetric key, obtained through the Diffie-Hellman key exchange protocol, to encrypt the content, then encrypts the symmetric key with the receiver IBE public key. IBE private key is loaded and encrypted to the phone SIM card at subscription time.

The authors give two examples of how Porscha behaves on a mobile phone: first using SM-S/MMS, and second using email. For the former, Porscha triggers the download of the content, when is finished, extracts the policy and, together with the Activity Manager, determines which applications can see the content. Every application compliant with the policies receives the content. For the latter, as Android email client applications use diverse protocols, such as POP, IMAP, or Active Sync, the email enforcement starts at the middleware level with the interception of email traffic. Then, Porsha interprets the email and checks if the email contains an email attachment. If the attachment is recognized as a policy associated, Porscha and the Activity Manager determines which email clients satisfy the policy and can retrieve the content.

Furthermore, Porscha also enforces content policy when content is stored or shared by applications on the mobile device. By default, Android stores these contents using Content Provider components. Every Content Provider record has an extra policy field in its structure that is used to save the policy associated to the content. The saved policy is checked every time the Content Provider record is accessed. This also ensures that other applications, with access to the same Content Provider, when trying to access a policy enforced content, have to satisfy the policy conditions to retrieve the content. Every time an application passes content, using an Intent, invoking an API call, or answering to one, the policy belonging is attached to it and is enforced by Porscha with Binder.

In summary, in our case neither OMA DRM nor Porscha could solve the problem of secure self-destruct messaging entirely. OMA DRM is too limited in terms of protections offered and Porscha is limited to mobile phones as it depends on the phone number of the device, and also doesn’t update the policy after this has been downloaded.
2.3 Summary

In this chapter, we have further motivated the need to provide adequate operating system support for self-destructing messages in mobile apps. Taking Snapchat as a representative example of such applications, we have seen from Snapchat’s history how challenging it is to enforce remote enforcement of self-destructing message policies. This difficulty stems partly from the complexity to program cryptographic protocols, and partly from the limitations of existing mobile operating systems, which, e.g., fail to block capturing printscreens when sensitive data is being displayed and are unable to provide secure data deletion on the remote party’s devices.

By analyzing the related work, we see that there is a lot of relevant work in mobile security, but none of existing solutions serves entirely our requirements. Information Flow Control enable users to keep track of how applications make use of their data, but provides no mechanisms that prevent the local user from overriding its protections using local operating system functions (e.g., screen capture). Existing access control mechanisms are focused primarily on supervising access to a given resource or data item, but are limited with respect to restricting how the data can be further used by the applications. Secure data deletion techniques are fundamental to ensure that residual sensitive data cannot be found on a given mobile device. Such residual data could be stolen by an attacker and violate data confidentiality requirements. Existing techniques, however, constitute only a building block in order to provide an end-to-end remote self-destructing message service. Digital Rights Management are the class of techniques that mostly closely relate to our problem. In particular, Porscha is a content protection framework that allows data owners to express security policies to ensure that their data is sent to targeted phones, processed by endorsed applications, and handled in intended ways. However, Porscha’s content policies target solely the protection of SMS, MMS, and email documents, rendering it useless to handle and protect images. Therefore, the research question we aim to address is: how can we design a middleware system that can help application developers enforce remote self-
destructing messages on mobile devices. The next chapter presents a solution to this question.
This chapter presents ShareIff, a sticky policy middleware to support self-destruct messages in Android applications. We start by providing a high-level description of the ShareIff architecture (Section 3.1) and then dive into its design details. In particular, we introduce its novel programming abstraction for embedding protected digital content on self-destructing messaging applications (Section 3.2), present tailored-made security protocols based on sticky-policies (Section 3.3), and describe the design of a set of Android extensions responsible for the enforcement of sticky policies (Section 3.4), with particular emphasis to the rendering engine which ensures that images are securely displayed on a remote party (Section 3.5).

### 3.1 Architecture

![Architecture of the ShareIff middleware.](image)

Figure 3.1 represents the architecture of the ShareIff middleware on an Android device. ShareIff consists of three main components: API, manager, and renderer components. The ShareIff API provides applications an interface to create envelopes on the sender (close operation), render them on the receiver (open operation), and manage ShareIff-specific keys. The ShareIff manager and the ShareIff renderer are components that reside in the OS domain and are isolated from user applications. The ShareIff manager holds sensitive cryptographic keys and performs encryption and decryption securely from the app. The ShareIff renderer displays in-
ternally decrypted envelopes on the local screen. At a high-level the typical workflow performed by application programmers when using ShareIff is as follows:

1. **Share public keys with other users.** Every ShareIff device contains a set of cryptographic keys upon which envelope close and open operations are built. In order for a user to be able to decrypt envelopes and visualize them on her device, the application must obtain the public keys stored by ShareIff on her device and share such keys with their friends. Obtaining the public keys is performed through an API call.

2. **Close envelope on the sender endpoint.** Once in possession of the ShareIff public keys of their friends, a user can send a self-destructing message securely to a friend by generating an envelope. The application invokes an API call to “close” an envelope containing the message content and a sticky policy that specifies the visualization timeout. The resulting envelope can then be forwarded by the application to the receiver, e.g., through an application-specific server (see Figure 2.1).

3. **Open envelope on the receiver endpoint.** When the application running on the recipient’s device retrieves the envelope, the enclosed message can be shown on the screen by invoking the “open envelope” API call. This call forwards the envelope to ShareIff’s protected components residing in the OS. ShareIff decrypts the envelope using the keys maintained by the manager and renders the recovered message (typically an image) on screen for the time duration specified in the envelope’s sticky policy. While the message is on display all potentially dangerous channels are blocked (e.g., screenshots are disabled). ShareIff provides that the message vanishes once the maximum time is enforced, that it is securely erased, and that the message cannot be opened twice, therefore avoiding message replays.

### 3.2 The Trustview API Abstraction

Among all ShareIff API calls, the open envelope operation is the most disruptive for application programmers since it affects the way they typically manipulate images or text messages on their applications. In fact, typical applications have full control of the images to display and their placement on screen. However, on the recipient’s endpoint, ShareIff applications have no access to the image enclosed in the envelop since the envelope is decrypted and rendered exclusively by the OS-protected ShareIff components.

To lessen the impact of such restrictions to application programmers, ShareIff provides a simple abstraction that allows for the manipulation of enveloped images in the application. This abstraction, named TrustView, is inspired on Android’s View class, which application developers are familiarized with. Typically, an application’s GUI is arranged into screens that contain UI
elements, such as buttons, text panels, etc. In Android, GUI elements and layout arrangements are defined by instances of subclasses of View. In particular, images are typically rendered inside ImageView objects, which represents a screen area in which images are displayed.

Trustviews borrow the concept of view in order to represent GUI elements where self-destructing messages can be securely rendered and deleted once the associated timeout expires. Such GUI elements are represented by TrustView objects and can be declared like a regular view, i.e., in the XML layout file or programmatically in the application code. Furthermore, trustviews can be incorporated into a typical view hierarchy, co-existing with regular views, just like in the example shown in Figure 3.2. The right hand side of the figure represents an application’s screen comprising: one regular view showing the application name (1), a trustview where a self-destructing message is shown (2), and two other regular views for the buttons “Like” (3) and “Back” (4). Listing 3.1 sketches how this layout hierarchy can be represented in XML, and Listing 3.2 provides the Java code to open an envelope inside the trustview instance.

In contrast to regular ImageView objects, trustviews have a hardened method interface. First, a programmer can only load content into a trustview so long as it is encoded inside an
CHAPTER 3. PROPOSED SOLUTION

Envelope envelope = getEnvelopeFromServer();
TrustedImageView tiv = (TrustedImageView)
    findViewById(R.id.tiv);
tiv.setImageBitmapFromEnvelope(envelope);

Listing 3.2: Java code to open an envelope in a trustview.

envelope. Second, the programmer cannot read or change the raw bytes of the trustview content. Furthermore, the programmer cannot change the visibility timeout associated with the content and specified in the envelope’s sticky policy. The programmer is restricted to a few method calls to: create or destroy a trusted view, configure its layout properties, and open envelopes in it.

3.3 Security Protocols

This section describes the security protocols that are triggered by client applications through the ShareIff API. We use the following notation to describe cryptographic operations. For asymmetric cryptography, \( K^- \) and \( K^+ \) denote private and public key, respectively. To refer to symmetric keys and asymmetric keypairs, we drop the superscript. Notation \( \langle x \rangle_K \) indicates that a piece of data \( x \) is encrypted with key \( K \) and \( \{x\}_K \) that \( x \) is signed with \( K \). The symbol || represents concatenation. Function \( \text{hmac}(m, k) \) yields the HMAC of message \( m \) with key \( k \).

3.3.1 API Primitives

As mentioned in Section 3.1, the ShareIff API provides a set of security primitives to perform three main functions: close envelopes, open envelopes, and manage internal cryptographic keys. Such keys are maintained by the ShareIff manager in each device and consist of three keypairs: platform key \( (KP) \), screening key \( (KS) \), and receiver key \( (KR) \). The purpose of these keys is clarified in the following sections. The core API primitives are:

\[
\text{CloseEnv}(M, T, C_{KP}, C_{KS}, [C_{KR}]) \rightarrow E_{[4|5]} : \text{Generates an envelope at the sender side. It takes a message } M \text{ to be sent, a maximum visualization time } T \text{, a certificate of the platform key } C_{KP}, \text{a certificate of the screening key } C_{KS}, \text{and (optionally) a certificate of the receiver key } C_{KR}. \text{These certificates must be obtained at the receiver’s endpoint through the local invocation of key management primitives. The return is an envelope, which can be of format } E_4 \text{ or } E_5 \text{ depending on whether or not } C_{KR} \text{ is provided as input (see Sections 3.3.5 and 3.3.6).}
\]

\[
\text{OpenEnv}(E, v) \rightarrow \text{OK | Fail} : \text{Renders the content of a received envelope } E \text{ on the trustview } v. \text{ If the call is authorized, ShareIff decrypts the message, shows the resulting image}
\]
3.3. SECURITY PROTOCOLS

according to the enclosed timeframe restriction, and returns OK. Otherwise, the message is not shown and the call aborts.

\texttt{GETPKEY()} \rightarrow C_{KP} : \text{Returns the certificate of the local platform key } K_P \text{ (see Section 3.3.3).}

The certificate must be transmitted to potential senders.

\texttt{NEWSKY}(C) \rightarrow C_{KS} : \text{Creates a new screening key on the local ShareIff service. The created key has a maximum envelope opening count } C. \text{ Returns a certificate of the screening key (see Section 3.3.5). The certificate must be transmitted to potential senders.}

\texttt{NEWRKEY()} \rightarrow C_{KR} : \text{Creates a new receiver key on the local ShareIff service. Returns a certificate of the newly created key } K_R \text{ (see Section 3.3.6). This certificate is only required by senders that require strong authentication of the message recipient.}

The ultimate goal of these primitives is to generate an envelope \( E \) (returned by \texttt{CLOSEENV}) that can be securely decrypted and rendered in the receiver’s device (when invoking by \texttt{OPENENV}). The generation of such an envelope is performed cryptographically and constitutes the main challenge since several security properties must be satisfied. To present the security protocols involved in the generation of ShareIff envelopes, we start from the simplest version \( E_1 \) to the most complete \( E_5 \).

3.3.2 Provide End-to-End Privacy

We start by ensuring end-to-end privacy between the sender and the receiver of a self-destructing message. To this end, the envelope is given by:

\[ E_1 \rightarrow (M, T)^{KE} \]

To generate \( E_1 \), ShareIff encrypts both the message \( M \) and a sticky policy \( T \) with an \textit{envelope key} \( KE \). The sticky policy includes the visualization timeout. The envelope key is a symmetric key generated at the sender side by \texttt{CLOSEENV}. For each envelope, there is a unique randomly-generated \( KE \). In order for the recipient to decrypt the envelope, the key needs to be securely transmitted to the recipient’s device. To prevent MITM attacks, the receiver’s client must be properly authenticated.

3.3.3 Bind to Remote Trustview Environments

The first step towards a proper authentication of the receiver’s endpoint is to bind the envelope to remote trustview environments. In other words, we need to guarantee that an envelope can be decrypted and rendered only inside a legitimate trustview environment protected
by the OS. Without such guarantee, the envelope can potentially be decrypted on the user space, allowing an application to save a permanent copy of the message, thereby violating the sticky policy.

\[
E_2 \rightarrow \langle M, T \rangle_{KE}, \langle KE \rangle_{KP^+}
\]

\[
C_{KP} \rightarrow \{ KP^+, ShareIff=KP \}_{KM^-}
\]

To bind the envelope to remote trustview environments, in \(E_2\), we encrypt the envelope key \(KE\) with the public key of a certified platform key \((KP)\). A platform key is a unique keypair that device manufacturers can deploy when bundling their ShareIff-enabled Android devices. Such a key can be stored in encrypted format and released to the ShareIff manager upon correct validation of integrity of the Android software during a verified boot sequence (Google 2015b). The public part of \(KP\) can be certified by the device manufacturer using its key \(KM\) so that the sender can validate whether that key corresponds to a legitimate ShareIff-enabled device. Since in such a device the private part of \(KP\) never leaves the ShareIff manager, by encrypting the envelope key with \(KP^+\), senders can be assured that the envelope can be decrypted only inside a trusted ShareIff manager service, which in turn restricts enveloped images to be rendered inside trustviews only.

### 3.3.4 Authenticate the Remote Application

A second step to properly authenticate the receiver’s endpoint consists of authenticating the application. In other words, we must ensure not only that the envelope is bound to be opened on trustviews only, but also that such trustviews belong to a trusted application. Otherwise, envelopes’ sensitive messages could be recovered inside trustviews attached to an illegitimate third-party application. To authenticate the remote application, we extend \(E_2\) as follows:

\[
E_3 \rightarrow \langle M, T \rangle_{KE}, id_A, hmac(m, KE), \langle KE \rangle_{KP^+}
\]

\[
m \rightarrow \langle M, T \rangle_{KE} \parallel id_A
\]

In this version, the envelope contains identity information \((id_A)\) about the targeted remote application. This identity can be obtained by simply calculating the hash of the application’s distribution package, which in turn is usually signed by the respective application programmer. When the ShareIff manager receives the envelope at the recipient endpoint, it first checks the identity of the local application that invoked the OPENEnv primitive and verifies if the identity of that application matches the identity specified in the envelope. If not, authentication fails and the envelope will not be decrypted. The envelope format \(E_3\) also includes an HMAC of the payload keyed with \(KE\) to guarantee the integrity of the envelope.
3.3. SECURITY PROTOCOLS

3.3.5 Prevent Replay Attacks

Before presenting the last step for complete remote peer authentication (see Section 3.3.6), we need to address the potential threat of replay attacks, in which a given envelope can be (partially or totally) visualized multiple times. The previous version of the protocol (E₃) is vulnerable to such attacks. On the one hand, an application can successfully invoke the OPENENV to open an envelope multiple times. Furthermore, as long as the platform key (KP) lives in the system, envelopes can be decrypted. This is because KP is used to decrypt the envelope keys. As a result, a leak of the platform key (for example, obtained through forensic extraction from persistent memory) may allow for the decryption of all past envelopes that have been sent to the device identified by that key.

In the next version of the protocol, we aim to prevent replay attacks such that an envelope can be opened a single time only and cannot be visible to the local user beyond the time frame indicated in the sticky policy. Thus, even if the application interrupts the visualization of the message before the maximum time frame, the receiver must not be able to replay the message. After finishing the view, the message must not be able to be recovered at all. For this purpose, we modify the protocol as follows:

\[ E₄ \rightarrow \langle M, T \rangle_{KE}, id_A, hmac(m, KE), \langle KE \rangle_{KS^+} \]

\[ C_{KS} \rightarrow \{ KS^+, ShareIff-KS, C \}_{KP^-} \]

In this version of the protocol, instead of encrypting the envelope key with the public part of the KP, the sender encrypts it with a screening key (KS). The screening key is in fact a keypair which is created by the ShareIff manager. To preserve the original link to the platform key, the public part of the screening key must be signed by the KP and the respective certificate sent to the sender. The sender can then verify the screening key certificate and generate the envelope as before but now using the KS to encrypt the envelope key.

There are two additional features about screening keys that mitigate the problem of the replay attacks. First, the screening keys are ephemeral. Let I be KP, KS, or KR, ShareIff preserves the private keys KI in volatile memory only and are never written to persistent memory. As a result, if the device is shut down or rebooted, KI keys are forcefully destroyed. Therefore, even if the KP is forensically extracted, it will not be possible to recover the private key KI that is required to decrypt the envelope key.

Second, the screening keys “have memory” of all envelopes that they have previously open. Whenever an envelope is requested to be open by an application, ShareIff maintains a volatile log associated with the screening key that records a hash of the envelope that was opened. If the same envelope is requested to be reopened, ShareIff checks the log and refuses to decrypt
CHAPTER 3. PROPOSED SOLUTION

Figure 3.3: Examples of screening sessions.

and render the image if an entry is found on the log, which can only be read and written by ShareIff. This method defeats repeated invocation attempts to open the same envelope.

To prevent the screening key logs from growing indefinitely, ShareIff sets a limit to number of different envelopes that can be open. In other words, screening keys expire after a pre-defined number of envelope open operations has been reached. This pre-defined number corresponds to the log capacity, which is the parameter $C$ of the $\text{NewSKey}$ operation (see Section 3.3.1). The log capacity $C$ is also included in the certificate returned by ShareIff upon creation of the screening key. Once the log is full, the key is destroyed by the system.

Adopting these techniques introduces some changes to the semantics of trusted view’s basic primitives. From the application point of view, this semantics can be captured by the notion of screening sessions. A screening session corresponds to the lifespan of a screening key $KS$. The session is opened when the key is created and closed if one of two events occurs: the device reboots, or the key expires. Envelopes can only be recovered when a session is opened.

We illustrate this notion in Figure 3.3. Figure 3.3 represents the lifespan of two sessions—A and B—initialized with a log size of 2 entries. This means that each session can open at most two envelopes. Five envelopes have been closed in total: $E^1_A$, $E^2_A$, $E^3_A$, $E^5_B$, and $E^3_B$ for session A, and $E^3_B$, and $E^5_B$ for session B. As shown by envelopes $E^1_A$ and $E^2_A$, an envelope can be opened only while its respective session is active (in this case session A). $E^3_B$ illustrates that an envelope can be opened once. Rebooting the device leads session A to termination. Opening a number of envelopes equal to the maximum log size causes session termination. This is what happens to session B after opening two envelopes: $E^4_B$, and $E^5_B$.

An immediate consequence is that some envelopes may never be able to be opened, either because the target device has rebooted and the screening key was lost or because the number of opened envelopes has expired. ShareIff applications must keep track of such situations and if necessary regenerate the envelope on the sender and send it again to the recipient.
3.3.6 Authenticate the Receiver

The last step to authenticate the remote endpoint is to authenticate the message recipient itself. Normally, in a typical messaging application, every user that signs up is assigned a unique name within the application domain. If someone wants to send a user a message, the recipient is identified based on this user name. For stronger security, ShareIff allows for the recipients to be authenticated directly using cryptographic mechanisms bundled into the envelope as follows:

\[ E_5 \rightarrow (M, T)_{K_E_1}, id_A, \text{hmac}(m, KE_1), \langle \langle KE_1 \rangle_{KS^+} \rangle_{KE_2}, \langle KE_2 \rangle_{KR^+} \]

\[ C_{KR} \rightarrow \{KR^+, \text{ShareIff-KR}\}_{KP^-} \]

Essentially, the idea is simply to encrypt the resulting encryption of \( KE_1 \) with \( KS^+ \) with another key (\( KE_2 \)) and encrypt \( KE_2 \) using a recipient key (\( KR \)), resulting in another layer of encryption using the \( KR \). The recipient key is a unique key associated to the local user. It can be generated by the application using the primitive \text{GenRKey} (see Section 3.3.1). This call returns a certificate of \( KR^+ \), but the respective private key never leaves the ShareIff manager. Thus, by encrypting the envelope key with the recipient’s public key, only the legitimate receiver can decrypt the envelope. This ensures strong recipient authentication. Since this last step is not crucial for the correctness of the self-destructing service, we leave it as an optional feature.

3.4 Policy Enforcement Architecture

Figure 3.4 gives us an overall perspective of the ShareIff policy enforcement architecture. As can be seen in this figure, when the receiver application receives an envelope containing the image to be displayed, it issues an OS system call (step 1) to the ShareIff Manager Service, so that it can process the image in question. This OS system call is transparently invoked by the \text{OpenEnv}.

At that point, this service extracts the envelope’s contents, validating its signature and other meta-data contained in the envelope, and decrypts the image (step 2). In possession of the image raw information, the service delivers this same information to ShareIffUI (step 3), a component created specifically to handle the rendering and posterior destruction of the image.

ShareIffUI corresponds to the ShareIff renderer and was designed to account and mitigate data leak channels present in Android’s stock architecture, such as the Android Device Bridge (ADB). In step 4, ShareIffUI addresses one of such vulnerabilities, namely the ability of an attacker to physically access the device using a debugger in order to retrieve the image in question. By disabling the communication between the OS and the (ADB) in the Systemw
CHAPTER 3. PROPOSED SOLUTION

Figure 3.4: ShareIff overall architecture.

Settings, ShareIffUI ensures that no physical attacker can plug the device to a computer and leverage ADB to either take screenshots (using the screencap command), or use a debugger to access variables that might contain the image’s raw information. ShareIffUI handles the screenshot issue in step 5. We created a flagging mechanism in Global Screenshot that allows ShareIffUI to temporarily disable the screenshot capability for as long as there is a privacy sensitive image being displayed by ShareIff. This capability can only be activated or deactivated by system level processes.

Then, Step 6 represents the actual rendering of the image, that is first invoked from the Window Manager, and traverses Android’s layers, until reaching the OS, where the rendering on screen actually takes place, already in the device driver. Section 3.5 provides additional details on the vulnerabilities addressed by this approach.

Finally, step 7, represents the image deletion mechanism. Basically, after issuing the render of the image, ShareIffUI launches a background thread that essentially works as a timer. After the timeout, ShareIffUI triggers the image removal through the Window Manager, signals the Global Screenshot to enable screenshots, and reestablishes the communication between the OS and ADB.
3.5 Rendering Engine

The rendering of privacy sensitive images in ShareIff is designed so that standard applications cannot access or modify these images. In a normal Android setting, an application developer is free to manage `ImageView` objects in his application’s activities, in order to display the images it desires, according to that application’s logic. In this work, because we want to prevent an application from manipulating these images, we added an additional mechanism that completely decouples the application UI logic from the rendering of these private images. Figure 3.5 gives us an overall perspective on how this rendering is done. In this figure we assume rendering is hardware-accelerated, a mechanism supported and generally used since Android 4.0. In Android, the rendering of an application screen is usually divided into two parts, the application UI, and the OS’s Status and Navigation bars. As we can see in Figure 3.5, the application’s UI is managed within that application’s process, while the Status and Navigation bars are managed by the OS’s SystemUI process. To ensure complete isolation between application logic and privacy sensitive images, we added a new component in the SystemUI process (represented in dark grey), that is effectively responsible for rendering these images. With this approach, whenever an application wishes to render a sensitive image, it can just issue an `openEnvelope` call to the operating system, effectively triggering the rendering of the image contained in the envelope, by the ShareIff renderer.
CHAPTER 3. PROPOSED SOLUTION

Given our threat model, this approach effectively mitigates the dangers associated with malicious applications trying to inadvertently access these privacy sensitive images. On the other hand, if we were to consider a compromised OS as part of our threat model, we would need to pay special attention to the structures that hold the image information, from the moment it is first extracted and decrypted from the envelope, to the time the image is actually rendered. In Android, the low-level rendering of images follows the producer-consumer architecture. In this particular case, the UI associated with the application and system bars are processed via OpenGL, and executed on the GPU, effectively making OpenGL an image stream producer. On the other hand, the Surface Flinger acts as an image stream consumer, as it is the component responsible for feeding these image streams to the lower Hardware Abstraction Layer (HAL), more specifically to the HWComposer and Display Controller, in order for the rendering to actually take place. Because image stream consumers and producers leverage Buffer Queue structures to perform their communication, one would essentially need to create a robust mechanism to protect these structures if we were to address the compromised OS scenario, something that falls outside this work’s scope.

3.6 Summary

This chapter presented ShareIff, a middleware that provides a service for enforcement of remote self-destructing messages for Android. Essentially, this system can be used to enhance the security of applications such as Snapchat. ShareIff’s design introduces several techniques. First, our system provides a simple programming abstraction named Trustview which enables developers to seamlessly embed protected content on the application’s screen. Trustviews are very similar to the kind of abstractions that developers are typically familiar with. Second, ShareIff includes a set of security protocols aimed to provide secure end-to-end delivery of protected messages. These protocols employ sticky policies in order to attach timeout restrictions to the visualization of images on remote parties. Third, ShareIff includes a set of design extensions to Android in order to securely render protected images on recipients’ devices. These extensions involve minimal changes to the Android operating system. Next, we provide more details about the implementation of the ShareIff system.
In this chapter, we present the implementation details of ShareIff, a sticky-policy middleware for the enforcement of remote self-destructing messages on Android. We start by providing a high-level overview of our implementation (Section 4.1), and then expand on more specific aspects of the implementation. In (Section 4.2) we explain the problem of handling big envelopes between processes in Android, the problem of rendering securely images and how we solved it in (Section 4.3), how we implemented our Sticky Policy API in (Section 4.4), and how we share keys between senders and receivers in (Section 4.6). Finally we present Photobrake in (Section 4.7), a simple Android application that uses ShareIff to send images between friends.

4.1 Implementation Overview

Our ShareIff prototype was developed on top of Android KitKat (4.4.1). We chose this version because of its maturity, but the process of porting ShareIff to a more recent Android version would be trivial, as the structures involved are present in the different versions. All the work involving end-to-end privacy and endpoint authentication follows the cryptographic protocols specified in Section 3.3. For symmetric cryptography we use AES-256, and for digital signatures RSA-2048 and SHA-2. Regarding key management, and because we opted to make the keys ephemeral, they are not stored persistently, but rather in volatile data structures that are destroyed once the device is turned off. All keys are generated using the default java security library.

Both keys and unprotected information are never stored persistently, but a more experienced user can examine the volatile memory content using the Android Debug Bridge (ADB). We managed to deactivate the ADB and the screenshot feature while the message content is present in the system in an unprotected form.

The evolution in camera technology made so that current smartphones are equipped with powerful cameras, which produce high resolution pictures, and therefore big files. One technical challenge we faced was the transmission of pictures and envelopes containing these big sized pictures between apps and the system service responsible for encryption / decryption. Because apps and system services reside in different processes, standard system service method calls are done through Android’s standard IPC mechanism (i.e., Binder). Given that this mechanism has
a fixed data buffer limit size of 1MB, we opted to handle pictures and envelopes by storing them persistently and using URIs to reference them. By doing so, we leveraged Android’s application sandbox mechanism to protect this data from other apps’ unauthorized accesses.

4.2 URIs to Handle Envelopes

When designing our ShareIff primitives, we had the goal of offering application developers an easy-to-use API. For instance, in order to create an envelope, ShareIff should offer a simple primitive, that would return a custom Envelope object, containing the usage policy associated with the image, as well as the image in encrypted form. However, the evolution in camera technology, led modern smartphones to be equipped with cameras that support resolutions of up to 41 MP. The higher the image resolution, the higher the image raw information size, which means that the Envelope object’s size, reflects the original image’s size.

To better understand this information size issue, we give an example based on the images we have chosen in our evaluation procedures. In order for an app to create an envelope containing a 5MP image, the app would need to send 4.3MB worth of image raw information to ShareIffManagerService. This means that there would be a need for transferring over 4MB of data between an application and an Android system service. Because applications and system services live in different processes, this communication needs to take place through Inter-process communication (IPC). Android offers a light and fast IPC mechanism called Binder, which is the default mechanism for communication between applications and system services. The problem however, is that this mechanism’s buffer has a limited fixed size of 1MB.

We solved the problem of passing the original image from the app to the service by passing an URI as argument of the envelope creation primitive instead. By doing so, we avoided forcing app developers to load the image from file, and made so that the service accesses the filesystem instead, and performs that same processing before the encryption phase. However, there is still a problem regarding the return value size of the primitive. For instance, the 5MP image given in the example, would yield an Envelope object whose size would be well above the 1MB fixed size limit of the Binder mechanism. To solve this issue, we made so that the service stores the envelope on file in the app’s private directory, and returns the corresponding envelope file URI. By doing so, we leveraged Android’s application sandbox mechanism to protect envelopes from unauthorized accesses from other apps.

On the other hand, we used this same URI approach for opening envelopes. When an app receives an envelope through the network, it leverages a primitive made available by a small middleware layer we created, to save the envelope in the app’s private directory. At that point, the primitive proceeds by calling the open envelope primitive of the ShareIffManagerService,
4.3 Secure rendering

The next issue was to securely render the image received by the application inside the envelope and passed to ShareIff, and make it look so simple as using an ImageView. When using an ImageView, we need to declare it in the application XML layout or instantiate it inside and activity and add it to the activity layout. An ImageView was not designed with the intention to show content protected by a sticky policy with its multiple configuration, neither to prevent the access to the image it contains.

The TrustedImageView extends the ImageView. It overrides the methods that allow a malicious user to access the information it contains and extends the super class by adding the setImageBitmapFromEnvelope method in order to receive an envelope and pass it to ShareIff. As explained in section 3.2, the programmer is very limited to what it can do with a TrustedImageView, since the methods from the super class were overridden to keep the content inaccessible.

A TrustedImageView is just an abstraction for the ShareIff logic that keeps the content safe from malicious applications. The image contained inside the envelope is never rendered in the application context. The TrustedImageView passes the envelope to the ShareIff trough the ShareIff API, using Binder, passing the envelope from the context of the application to the operative system context where it can be decrypted safely and reach the ShareIffUI component.

The ShareIffUI component implements the addShareIffContent method that receives an image and a policy, and the removeShareIffContent, called by the background thread when the time limit is reached. For the sake of simplicity we assume that no TrustedImageView was passed and ShareIff is going to use its own default fullscreen TrustedImageView to render the image, as it happens on Snapchat. The method addShareIffContent creates a TrustedImageView and configures its layouts parameters to fullscreen and overlap any existent screen being displayed. The policy is enforced before the image is sent to the Window Manager using the method addView.

During the time the image is shown on screen, the unencrypted version of the image resides on the operative system context, so any attempt to get a screenshot programmatically will result in a screenshot of the image of the application bellow the TrustedImageView. This is due to when trying to get a screenshot programmatically, Android traverses the application layout elements, recording the content of the elements. Since this TrustedImageView is not part of the layout of the application, its content is not going to be recorded. After the timeout defined by the policy, the method removeShareIffContent removes the TrustedImageView that was being
CHAPTER 4. IMPLEMENTATION

shown and resets all the system properties to the state they were before displaying the protected image.

4.4 ShareIff Policies

As explained in the previous section, the policy plays an important role while displaying the image and to allow application developers and application users to customize and have some control over the features that ShareIff provides, we provide an API to build and customize the sticky policy attached to message. ShareIff only accepts this policy, and if one is not provided it will use its standard policy, screenshots and ADB disabled, and image shown during 10 seconds.

It is important for a user to control the way the information is displayed on the receiver side, to know which options it has and not just use the standard definitions used by default. It is also important to provide this API for developers to be presented in their applications and provide transparency to the way their application uses ShareIff.

The ShareIff Policy is adaptable and scalable, and any improvements to it can be released with updates to the operating system. As it is today, the ShareIff policy allows the user to specify the timeout for the message and if it allows the receiver to take a screenshot while displaying the content.

adb shell screencap -p /<dest>/<dir>/picture.png
adb pull /<dest>/<dir>/picture.png
adb shell rm /<dest>/<dir>/picture.png

Listing 4.1: Screencap command via adb shell.

4.5 Screenshot Protection and ADB

In Android there are three ways to take a screenshot of the device screen. If the device is running Android 4.0 or later, one can press a key combination on the device and the operative system will take a screenshot of the entire screen, one may use the Android Debug Bridge \(^1\) to get a screenshot too, and finally one may code in the application a set of instructions to store an image that is being presented on the screen.

For each of these threats we developed protections to prevent screenshots being taken. For the first case, we studied how the process flows. After the keys are pressed, a new service is launched by the operative system in order to take a screenshot and save it to external memory.

\(^{1}\)http://developer.android.com/tools/help/adb.html
4.6. KEY MANAGEMENT

In case one presses the key combination in an attempt to launch the screenshot service while the device is showing protected content, ShareIff disables the Android screenshot mechanism by notifying the `WindowManagerService` that a ShareIff protected content is going to be displayed on screen.

The `WindowManagerService` makes the bridge between the `ShareIffUI` and the `WindowManagerPolicy`, the interface responsible for supplying all the UI-specific behavior of the window manager. The `PhoneWindowManager` implements this interface and the `interceptScreenshotChord` method, responsible for taking a screenshot of the screen. This way, when an attempt is made to take a screenshot and this mechanism was disabled via ShareIff, a message is presented on screen stating that protected content is being displayed.

As explained in the previous section, this mechanism can be controlled by the sender. If he wishes to allow the receiver to take screenshots of the content, the `ShareIffUI` will notify the `WindowManagerService` that a content is going to be displayed without the need of disabling the screenshot mechanism and the `PhoneWindowManager` will have its default behaviour.

For the second case a more experienced user may connect the device to a computer with ADB installed and get a screenshot using the commands in Listing 4.1 and not leave a trace that a screenshot was taken.

We decided to use a feature already present in Android that cannot be modified by a user application. The `ADB_ENABLED` constant present in `Settings.Global` can only be modified by the operating system or by a system application and is used to enable or disable the ADB in the device. Since the `ShareIffUI` runs within the operating system context, we can modify this constant to disable the ADB before a protected content is displayed and enable it after. The user on the computer connected to the device has no means to modify this constant.

For the last one, one could try to create a new Bitmap from the drawing cache of the root view of the window being displayed on screen. With our approach of taking the drawing of the protected content from the application context and passing it to operative system context, we prevent an application from getting data from the protected content due to properties of memory isolation of Android. We also took additional measures, we extended the class responsible for drawing application images on screen and overrode the methods to prevent any access to the data being displayed.

4.6 Key Management

Another key issue for application developers and application users are the keys used to encrypt and sign messages. As explained in section 3.3.1, the $C_{KP}$, $C_{KS}$, and $C_{KR}$ must be generated, kept and securely transmitted to potential senders. Section 3.3.3 proposes a solution
CHAPTER 4. IMPLEMENTATION

Figure 4.1: Sharing keys between Photobrake users.

to generate and keep the $KP$ in the device. To complement we assume that $C_{KM}$ is a trusted well known certificate that is included in the ShareIff-enable Android OS, signed by a Trusted Root CA, and can be used to verify the received $C_{KP}$ before opening the envelope. The $C_{KS}$ and $C_{KR}$ require a different solution to be securely transmitted, as presented next.

Since the generated keys are related to applications, ShareIff keeps the records of this relations, and the certificates of the public keys are known to the application, we decided that it should be the application to handle how the keys are shared between users. In Figure 4.1 we present a simple example of how the $C_{KS}$ and $C_{KR}$ can be shared using the Photobrake application (Section 4.7), whose key management architecture could be adopted by other applications. The Photobrake servers are trusted and are responsible for managing the user database and forward information shared between users. The clients are instances of Photobrake application running on ShareIff-enable Android smartphones. When launching the application, if it runs for the first time, it will get the $C_{KP}$, $C_{KS}$ and $C_{KR}$ using ShareIff API (2) and register them on the Photobrake servers along with the user identification (3). If $C_{KS}$ has expired, it will get a new $C_{KS}$ and Photobrake registers the new certificate in the server. When a friend adds another friend (4), after accepting the request (5), both friends receive the key certificates from each other (6) making the application responsible for keeping the certificates. This way, both clients have the keys necessary to send ShareIff envelopes to each other.

In the next section we provide more details about Photobrake and how it uses ShareIff to send images and safely render images.
4.7. **PHOTOBRAKE**

To properly demonstrate the above functionalities and how easy it is to use the ShareIff API, we implemented Photobrake. This application provides a self-destruct messaging service similar to those of Snapchat, i.e., it allows users to share photos with specific visualization time frame policies associated to them. Essentially, when sending a photo, the user is presented with an activity that allows for selecting the photo, picking the receiver from a friend list, and specify the timeout. These policies will then be enforced by ShareIff when they arrive to the receiver’s device (e.g., a photo sent by Photobrake, can only be opened by the same application on the receiver’s device, and the receiver-side screenshot service prevents the user from acquiring the photo). Similarly to Snapchat, Photobrake relies on a centralized server to forward snaps between clients.

To use Photobrake, the user needs to login into the application. After that, the user can see the images he received from his friends or send a picture. To send a picture, the user can take a picture using the device’s camera or choose one already saved. After choosing the picture, a new screen is presented to him with the options of the sticky policy that is going to be added to the picture and enforced by ShareIff on the receiver. Finally he chooses the friends he wants to send the picture and presses send. During all this process the user is never aware that he is using ShareIff such as the application programmer that is abstracted from its implementation. A sample of the code used by Photobrake to send a message can be seen on Listing 4.2.

Photobrake takes advantage of the default implementation of ShareIff. If a trustview doesn’t exist, ShareIff will create a fullscreen trustview and display the content inside. To do so we instantiate a `ShareIff` object to provide the API and we pass the `Envelope` object obtained from the Photobrake server to the `openEnvelope` method as shown on Listing 4.3.
4.8 Summary

In this chapter, we presented our implementation of the ShareIff system. ShareIff was built for the Android platform (KitKat version). In order to implement the necessary operating system extensions required by ShareIff, it was necessary to modify the Android source code. In spite of that, the code that was modified was very limited. To perform these low level source code edits, it was necessary to overcome several challenges related with limitations in internal buffers which complicated the communication between the application’s user space and privileged Android services. We also presented Photobrake and how it uses ShareIff to share images. A proper evaluation of our system is conducted in the next chapter.
This chapter presents the evaluation of ShareIff. We evaluate our system in three complementary dimensions: performance overhead measurements (Section 5.1), a qualitative evaluation of its API (Section 5.2), and a security analysis of the system (Section 5.3).

5.1 Performance Evaluation

To study the performance of ShareIff, we measure the execution time of its API calls, through microbenchmarking. Our hardware testbed consisted of a Nexus 4 smartphone, featuring a quad-core 1.5 GHz CPU, 2 GB of RAM, 16 GB of memory, a 768 x 1280 display, and a camera with 8 MP, 3264 x 2448 pix. The device was flashed with a build of Android 4.4.1 AOSP patched with ShareIff code. For each experiment, we report mean and standard deviation of 50 runs.

This performance study features the execution time measurements of two types of operations: (1) creation of an envelope, and (2) opening of an envelope. The first type of operation features the encryption of an image, and the creation of an envelope structure containing that image and a custom ShareIff policy. For testing purposes, our custom policy is specified so that the image in question cannot be captured via screenshot while on the receiver’s device, and it also features a 10 second timeout, after which the image is deleted from the screen. The second type of operation encompasses the decryption of the image, as well as its rendering on a trusted image view. Although this second operation can be broken down to these two phases, atomically, it can just be seen as the primitive responsible for rendering an image using our ShareIff approach. For this reason, we decided to compare this second operation with Android’s original way of rendering an image.

Considering that the performance of these operations depend on the size of the images involved, we decided to make measurements with images with different sizes. For a matter of consistency, we decided to use the same image, and resize it to resolutions compatible with today’s mobile device cameras. Although today’s cameras support 16:9 resolutions, we decided to use a 4:3 image instead, in JPEG format, and use several of our Nexus 4 camera supported resolutions. When picking the resolutions we took under consideration Android’s original `setImageBitmap` method’s maximum resolution limits (i.e., 4096x4096 pixels). When discussing the results of these operations, we address the different images by their resolutions.
CHAPTER 5. EVALUATION

Figure 5.1: Execution time of envelope creation.

The details on the resolution and file sizes of the images in question can be found on Table 5.1.

<table>
<thead>
<tr>
<th>Resolutions (MP)</th>
<th>Resolutions (Pixels)</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3MP</td>
<td>640x480</td>
<td>200KB</td>
</tr>
<tr>
<td>2MP</td>
<td>1600x1200</td>
<td>1.4MB</td>
</tr>
<tr>
<td>3.1MP</td>
<td>2048x1536</td>
<td>2.6MB</td>
</tr>
<tr>
<td>5MP</td>
<td>2592x1944</td>
<td>4.3MB</td>
</tr>
<tr>
<td>8MP</td>
<td>3264x2448</td>
<td>6.8MB</td>
</tr>
<tr>
<td>12MP</td>
<td>4096x3072</td>
<td>10.9MB</td>
</tr>
</tbody>
</table>

Table 5.1: Experiment image resolutions and file size.

Figure 5.1 shows us the execution times of our envelope creation primitive, when handling different resolution images. In this figure we can observe a linear pattern in the overhead introduced by the image encryption operation, the most time consuming component of this primitive. While there is an increase in this primitive’s overhead, as the resolution of the image’s increases, we can also observe that the growth rate of the overhead progressively decreases.

Figure 5.2 shows us a comparison between the image rendering execution times on native Android, and on ShareIff. As we can see, for lower resolution images the overhead of ShareIff’s rendering can take close to twice as much when compared to Android’s native rendering. However, while the overhead increases logarithmically, it does so at a slow growing rate, as we can observe in the higher resolution images of this rendering process. The times presented for the ShareIff implementation include the AES and RSA decryption.

It is important to stress that, in Android, we can measure the time spent on rendering an
image by measuring the `setImageBitmap` method call’s execution time. However, in order to reliably compare ShareIff’s implementation with Android’s, it is also important to consider the execution time of the `BitmapFactory.decodeByteArray` method call as well. This is a computationally heavy method, that loads the data to be rendered from memory. Without accounting for this method, ShareIff’s overhead compared to Android’s native rendering execution would lead to great discrepancies. Therefore, our Android native rendering measurements account the execution times of both the decoding operation, and the actual rendering method call.

If we look at Figure 5.2 we can see that for 12MP images, the rendering time is of more than 3 seconds, which may compromise the user’s experience. Even so, Android’s native rendering for 12MP images is only half a second faster, which in comparison is not a great improvement. To preserve the user’s experience, Android usually uses thumbnails to alleviate this time difference. For instance, before displaying a large image in the gallery, Android usually presents a lower resolution version of the same image (i.e., thumbnail), and then presents some sort of loading screen before showing the real image.

### 5.2 Case Study

In order to test the effectiveness and benefits of the functionalities provided by ShareIff, we developed a use case application called Photobrake.
Building the Photobrake application allowed us to gather hands-on experience about the programming complexity of ShareIff. Regarding both the creation of new envelopes and their respective decoding, ShareIff’s primitives are very concise and intuitive. In less than 10 lines of code, it is possible to write the necessary Java code to create an envelope. Similarly, opening an envelope entails also just a few lines of code in order to invoke a single trustview method.

However, the API methods involved in the management of the ShareIff-specific keys introduced an additional complexity when compared to a regular application like Snapchat. First, users must explicitly upload platform key certificates to the server the first time they register on the application. Second, Photobrake clients must necessarily create a new screening key every time the device reboots (remember that screening sessions are stored in volatile memory) and the respective certificate must be uploaded to the server. Third, the application must keep track of the log count associated with each screening session and if necessary generate a new screening key. Fourth, the application must keep track of snaps that could not be viewed due to a reboot of the recipient device. Nevertheless, we found that most of these steps could be performed relatively easy by extending the server API and adding a few records to the server database to keep track of the state of each key.

5.3 Security Analysis

Except for the “analog hole” limitation (L1), the ShareIff middleware can effectively overcome the security limitations of Snapchat discussed in Section 2.1.2. ShareIff thwarts limitation L2, by disabling the screenshot and ADB functions while snaps are displayed on screen. Limitations L3, L4, and L5 are overcome by ensuring secure end-to-end encryption of snaps’ data between the sender and receiver and by the fact that raw snap data can only be accessible within the operating system. Replay attacks (L6) are defeated by employing volatile keys (screening keys) for encrypting the envelopes. Lastly, L7 is mitigated by employing verified boot mechanisms in order to detect rooted Android software.

If the phone gets stolen and the attacker turns off the phone, the volatile keys are automatically destroyed. By offering a rendering mechanism decoupled from the app, we can ensure that no untrusted application can overlay an image of its own covering the envelope’s image. The image removal mechanism resides in the OS, which means no app can remove the image before the time is up (except through the back button), or let the image on screen for more time than the policy first stipulated.

In the current ShareIff design, because the creation of an envelope involves specifying the receiver’s id through it public key, there is a dependency from the system service and an external service that translates a known user id into a certificate. It is from that certificate that the service extracts the receiver’s public key to proceed with the encryption the envelope’s contents.
5.4 Summary

In this chapter, we presented our evaluation of the ShareIff system. Our main findings from our experimental evaluation are that the performance overhead introduced to both the operating system and to applications is negligible. To qualitatively evaluate ShareIff, we developed a simple use case photo sharing application named PhotoBrake. From our experience in developing this app we found that, although ShareIff introduces some complexity in the management of cryptographic keys, embedding trustviews in the application code is quite straightforward. Finally, from our security analysis, we conclude that ShareIff manages to mitigate all attacks that were contemplated in our threat model. The next chapter finishes this thesis by presenting the conclusions and also discussing some directions in terms of future work.
In this dissertation we presented ShareIff, a middleware for Android that provides an API for secure sharing and display of self-destructing messages. Many attempts have been made to create such messaging services, with Snapchat being perhaps the one who gained more popularity. Even so, although not suffering severe repercussions, even Snapchat suffered from limitations of current mobile operating systems that compromised its message secure display and deletion principles. In this work, we seek to tackle both the end-to-end security when exchanging messages, but also the secure handling of messages on the receiver’s device. ShareIff offers apps the possibility to encrypt a message on the sender’s endpoint and send it to the recipient such that the message can be decrypted and securely displayed only on the recipient’s device for the amount of time specified by the sender. ShareIff does so by offering application developers a programming abstraction called TrustedView, that introduces marginal overheads to both system and application.

From the work that was perform, we highlight three main conclusions. First, by providing adequate middleware support, it is possible to enhance the security of mobile applications such as Snapchat, which offer to their users some form of remote control over shared data. The foundations that were used to remotely enforce such kind of control are rooted on a combined hardware / software trusted platform assembled by the device manufacturer. So long as the trusted platform preserves its integrity, users are assured that enveloped data that they share over ShareIff remains visible on the target remote device for only the amount of time that was specified by the original user. Second, exposing this functionality to the application programmer can be done without significantly increasing the programming complexity. ShareIff provides the trustview abstraction which makes the integration of sensitive image display into the application’s screens to be intuitive for programmers. Finally, extending Android to be ShareIff-compatible can be carried out without (1) deep changes to the operating system and (2) impairing the performance of the operating system and applications. As a result, ShareIff poses relatively low barriers for deployment in the real world.
6.2 Future Work

As for future work, we envision the following potential topics of research:

1. **Increase expressiveness of sticky policies.** In the current version of ShareIff, users can specify only simple policies that restrict the maximum amount of time that a given image can be shown to the user on the recipient’s device. We envision that such policies can be much more expressive so that they can accommodate a richer restrictions, for example, support content “re-sharing” with other users, allow for more expressive policies involving time (e.g., constrain the visualization of an image to be performed within a particular time window), support policy restrictions based on location information, etc. In order to accommodate richer policies, ShareIff needs to provide additional security mechanisms to allow secure delegation of envelopes with other users, implement trusted time sources, and reliably determine location. In addition to the design of such mechanisms, it is necessary to determine the impact they will bring to the programmability of applications and to the overall performance of the system.

2. **Reduce the Trusted Computing Base.** As of now, the Trusted Computing Base (TCB) of the system is considerably large. In particular, it consists of the entire Android system which entails millions of lines of code. As a result, the potential of outstanding vulnerabilities in the system code is rather high, and therefore more prone to attack than in systems will smaller TCBs. A potential research direction is to reduce the TCB size of the system by isolating the critical ShareIff code responsible for decrypting and rendering sensitive content on screen on an security domain isolated from the Android operating system. Therefore, even if Android has been compromised, an attacker could not have direct access to sensitive envelope data since it would be inaccessibly protected by an independent security domain. To implement such an isolated security domain, a possible avenue for exploration is to leverage ARM TrustZone technology. An alternative approach is to explore the usage of trusted hypervisors to isolate Android from the security domain.

3. **Scalable cryptography.** As is, ShareIff requires the same information to be encrypted with different keys, corresponding each key to a receiver. For instance if a user is sending a large chunk of data to a large group of friends, it will have to encrypt the same chunk to each one of her friends, causing the overhead of the encryption to be multiplied by the number of users and causing a huge usage of mobile traffic data since the application using ShareIff will have to send each envelope. It would be interesting to have a way to encrypt the data only once and make it available to a group of users. This problem is related to the problem of broadcast encryption and applying the existent solutions could be a path to follow. Other approach could be to apply attribute-based encryption in order to make the information available only to a specific group of people that have an attribute in common, like a user defined group.


